DAA/LANGLEY

NAG1-613

Report No. UIUCDCS-R-87-1327

UILU-ENG-87-1714

A FUNCTIONAL PROGRAMMING INTERPRETER

IN-61
64476-CR
p65

by

Arch Douglas Robison

March 1987

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

(NASA-CR-180618) A FUNCTIONAL PROGRAMMING INTERPRETER M.S. Thesis (Illinois Univ.) 65 p Avail: NTIS HC A04/MF A01 CSCL 09B

N87-26516

Unclas

G3/61 0064476

A FUNCTIONAL PROGRAMMING INTERPRETER

by
Arch Douglas Robison

THESIS

DEPARTMENT OF COMPUTER SCIENCE
1304 W. SPRINGFIELD AVENUE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, IL 61801

This work was supported by a grant from NASA
Contract Number NAG 1-613

A FUNCTIONAL PROGRAMMING INTERPRETER

BY

ARCH DOUGLAS ROBISON

B.S., Case Western Reserve University, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Roy Campbell for support and suggestions during the project. I thank all the members of the Professional Workstation Research Group for answering my questions about the UNIX anatomy. The Computer Science Department and CRL are to be commended for providing the excellent computing facilities which made this project possible. Finally, I would like to thank John Backus for taking time to give the lecture which inspired this thesis.

TABLE OF CONTENTS

CHAPTER 1.

Introduction

Functional Programming (FP)[Bac78] is an alternative to conventional imperative programming languages. This thesis describes an FP interpreter implementation. Superficially, FP appears to be a simple, but very inefficient language. Its simplicity, however, allows it to be interpreted quickly. Much of the inefficiency can be removed by simple interpreter techniques.

This thesis details the design of the Illinois Functional Programming (IFP) interpreter, an interactive functional programming implementation which runs under both MS-DOS and UNIX.[1] The IFP interpreter allows functions to be created, executed, and debugged in an environment very similar to UNIX. IFP's speed is competitive with other interpreted languages such as BASIC.

[1]MS-DOS and UNIX are trademarks of the Microsoft Corporation and Bell Laboratories respectively.

CHAPTER 2.

Background

"Look," said Roark. "The famous flutings on the famous columns —what are they there for? To hide the joints in wood—when columns were made of wood, only these aren't, they're marble. The triglyphs, what are they? *Wood.* Wooden beams, the way they had to be laid when people began to build wooden shacks. Your Greeks took marble and they made copies of their wooden structures out of it, because others had done it that way. Then your masters of the Renaissance came along and made copies in plaster of copies in marble of copies in wood. Now here we are, making copies in steel and concrete of copies in plaster of copies in marble of copies in wood. Why?"

- Ayn Rand, *The Fountainhead*

2.1. Origins

Conventional computer languages descend directly from machine language. Each language feature traces its ancestry to a machine level counterpart. Variables evolved from storage locations; GOTO and IF statements mimic jump and branch instructions. Though these abstractions of machine language free the programmer from machine- dependent detail, they still force the programmer to work on the same word-at-a-time level as the machine.

In his 1978 Turing Award Lecture, John Backus presented a critique of conventional languages and an alternative: functional programming (FP).[1] FP programs have neither the control flow nor variables of conventional languages. Instead programs are directly constructed from smaller programs.

Berkeley UNIX has an FP implementation[Bad83]. Berkeley FP follows Backus' FP definition closely, except for minor character set changes. The Berkeley FP interpreter is

[1]From here on, FP will denote any functional programming employing the combinator style of Backus' FP. Backus' FP will always be referenced explicitly as such.

written in Franz LISP. Functions are translated into lisp functions which are then interpreted by the LISP interpreter. Alternatively, the LISP functions may be compiled into machine code.

The Berkeley FP is slow; a Berkeley FP program may run as much as 20 times slower than its BASIC counterpart. The interpreter carry's the unnecessary overhead of Franz LISP. The simplicity of FP is lost when translated into LISP. For example, FP has a trivial calling convention: every function has a single anonymous argument. In contrast, the LISP translation uses lambda-bindings with multiple arguments, which are more complex and time-consuming. A simple language should be evaluated by simple methods.

2.2. Goals

Most programming languages are extremely complex, thus making mere implementation complex and difficult, much more so an efficient implementation. FP is a much simpler language, so more effort could be made towards an efficient and friendly implementation.

There were three general criteria for the interpreter. First, it should include debugging features such as a trace mechanism and domain error messages. Second, the interpreter should run on generally available computer systems. Finally, it should run quickly. Most available computers are of the Von-Neumann model, so the interpreter must run efficiently within the constraints of the "Von-Neumann Bottleneck".

The interpreter was specified to have the following features:

1. Define new FP functions.
2. Evaluate an FP application.
3. Trace the evaluation of an application.

4. Show the reasons (back-trace) for an undefined result.

The first two features simply implement Backus' FP system. The tracing feature was adopted from Berkeley FP. The last feature was specified from experience with Berkeley FP. Errors in FP programs result in a undefined value which propagates to the final output. Within Berkeley FP, there is no reporting of why an error occurred, so debugging programs is extremely difficult. Though the error reporting is not pure FP (the error message is a side effect!), it is of great practical value.

CHAPTER 3.

Language

This chapter gives an overview of the IFP language. The reader is referred to the user's manual[Rob85] for the finer details such as where to put the semicolons.

The IFP domain includes *objects, functions,* and *program forming operations* (PFO). PFO are sometimes called *functional forms*. They correspond to data, procedures, and control structures in conventional languages respectively.

3.1. Objects

The objects of IFP represent data. Objects may be atoms, sequences, or bottom. Atoms are scalar data such as numbers, boolean values, and strings. A sequence corresponds to LISP's list. *Bottom* is neither atom nor sequence, it denotes an undefined result.

3.1.1. Bottom

The simplest IFP object is *bottom,* which is written in IFP as "?". *Bottom represents an undefined value, such as the result of division by zero.* Like Backus' FP, IFP has the *bottom preserving* property. The bottom preserving property requires that application of a function to an undefined input results in an undefined output. A corollary is that any data structure containing "?" is itself equal to "?".

3.1.2. Atoms

In Backus' FP there is only one type of atom, the string. Some strings represent numbers or boolean values. Arithmetic functions could be applied to numeric strings, as in *awk*Aho84 but there are both theoretic and practical reasons for not doing this. On the practical side, arithmetic on the string representations of numbers is slow on most machines. A clever interpreter, however, could store numeric strings internally as integers or floating point numbers.

There would still be a theoretic problem. Consider the string "0.000". If stored as a floating point number, it would be indistinguishable from the string "0.0". When treated as strings we wish to distinguish between the two, when treated as numbers we want the two strings to be considered equal. We could have both "string equality" and "numeric equality" tests, but this would seem unnecessarily complex. The IFP atomic domain therefore contains the mutually exclusive types of character strings, integers, reals, and boolean values. The IFP type partitioning is shown in figure 1. The boolean constants "true" and "false" are denoted *t* and *f* respectively.

3.1.3. Sequences

An IFP sequence is a tuple of objects delimited by angle brackets. Below are some sequences:

```
<a b c>
<3 1 4 1 5>
<>
<plus <a b> 2>
```

In Backus' FP, empty sequences are also atoms; in IFP empty sequences are not atoms. The choice is somewhat arbitrary, it mostly emphasizes a viewpoint. In LISP, the empty list is an

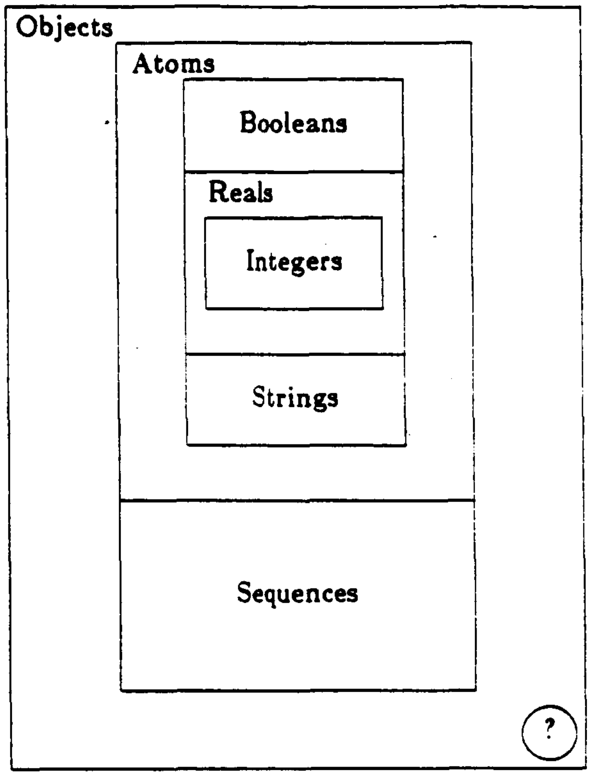

Figure 1

atom since it is a leaf of LISP's tree structures. In IFP, the sequences are more like arrays, so the empty list is treated as an empty array.

3.2. Functions

IFP functions map objects into objects. They are true functions in the mathematical sense in that the mapping is injective. IFP functions preserve referential transparency[Ten76], that is the identity:

$$f(x)=f(x)$$

always holds. IFP functions are strict and bottom preserving[Bac78]. We denote the application of a function to an object with a colon, that is:[1]

$$x:f \rightarrow f(x)$$

The terminology should be clarified at this point. The string denoting the application of *function* f to *object* x is called an expression. Expressions are distinct from functions and objects.

3.2.1. Primitive Functions

Primitive functions are built into the interpreter. Since they are too numerous to all be listed here, only those used in this thesis will be described. Most mathematical functions have their usual names in IFP. For example, the sine and cosine functions are *sin* and *cos*. Dyadic functions such as addition expect a two-element sequence (pair) as input. For example:

$$<3\ 4> : + \rightarrow 7$$

Multiplication and division are denoted by "*" and "%" respectively.

The identity function *id* returns its argument.

$$x:\text{id} \equiv x$$

The *null* function tests if a sequence is empty.

$$<>:\text{null} \equiv t <x_1, \cdots>:\text{null} \equiv f$$

The *apndl* and *apndr* functions append an element to the left and right sides of a sequence respectively:

[1]To minimize confusion, we use two different symbols for equality. The symbol $\equiv$ indicates that the left and right sides are both IFP expressions. The symbol $\rightarrow$ indicates that the left side is IFP and the right side is standard mathematical notation.

$$<x,<y_1,y_2,\cdots y_n>> : \text{apndl} \equiv <x,y_1,y_2,\cdots y_n>$$

$$<<x_1,x_2,\cdots x_m>,y> : \text{apndr} \equiv <x_1,x_2,\cdots x_m,y>$$

The *tl* and *tlr* drop the first or last element from a sequence:

$$<x_1,x_2,\cdots x_n> : \text{tl} \equiv <x_2,\cdots x_n>$$

$$<x_1,x_2,\cdots x_n> : \text{tlr} \equiv <x_1,x_2,\cdots x_{n-1}>$$

The *cat* function catenates sequences:

$$<<x_{11},x_{12},\cdots x_{1n_1}>,<x_{21},x_{22},\cdots x_{2n_2}>,\cdots <x_{m1},x_{m2},\cdots x_{mn_m}>> : \text{cat} \equiv$$
$$<x_{11},x_{12},\cdots x_{1n_1},x_{21},x_{22},\cdots x_{2n_2},\cdots x_{m1},x_{m2},\cdots x_{mn_m}>$$

There are also *structural* functions[Bac78] which replace the loop and subscript structures of conventional languages. The three principle structural functions are *distl, distr* and *trans,* which are defined below:

$$<x,<y_1,y_2,\cdots y_n>> : \text{distl} \equiv <<x,y_1><x,y_2>\cdots<x,y_n>>$$

$$<<x_1,x_2,\cdots x_n>,y> : \text{distr} \equiv <<x_1,y><x_2,y>\cdots<x_m,y>>$$

$$\begin{array}{llll} <<x_{11} & x_{12} & \cdots & x_{1n}> \\ <x_{21} & x_{22} & \cdots & x_{2n}> \\ \cdots & \cdots & \cdots & \cdots \\ <x_{m1} & x_{m2} & \cdots & x_{mn}>> \end{array} : \text{trans} \equiv \begin{array}{llll} <<x_{11} & x_{21} & \cdots & x_{m1}> \\ <x_{12} & x_{22} & \cdots & x_{m2}> \\ \cdots & \cdots & \cdots & \cdots \\ <x_{1n} & x_{2n} & \cdots & x_{mn}>> \end{array}$$

The heavy use of structural functions by IFP programs affects the interpreter design. As will be described in Chapters 4 and 6, the structural functions diminish interpretive overhead at the expense of creating many temporary storage structures (sequences) which must be allocated and deallocated quickly.

3.2.2. Function Organisation

Experience with Berkeley FP indicated that a single flat name space was a serious inconvenience, since the user would have no way to group functions in a logical manner. Therefore

IFP adopted a tree-structure system directly analogous to UNIX's file structure. Each function exists in *module* which corresponds to a UNIX directory. Functions are named in the same manner as UNIX files, either by a relative or absolute pathname. Furthermore, the user can establish the equivalent of symbolic links via an *import* file in each module. The hierarchical function organization is syntactic sugar. All pathnames are expanded to absolute form when parsed, so the IFP name space is lexically scoped.

Figure 2 shows part of a typical function hierarchy. The root node is labeled "r". The function "+" belongs to the module "arith", which contains arithmetic functions. The module "arith" in turn is a subdirectory of "math". The addition function would be referenced by "/math/arith/+". If it were imported into a given module, then "+" would suffice.

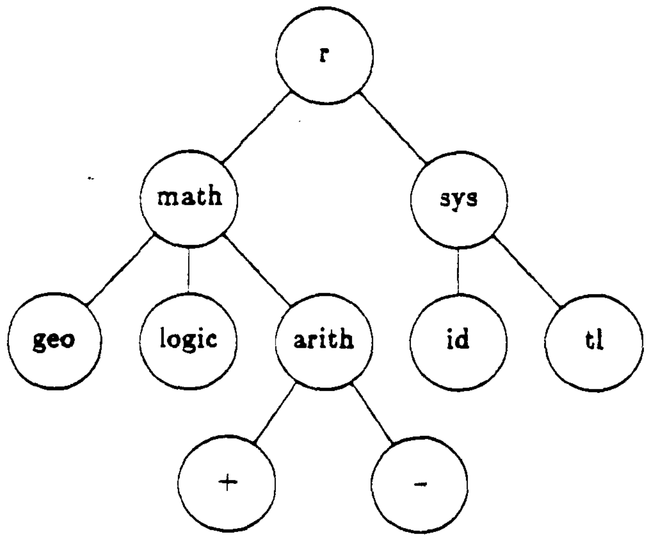

Figure 2

3.2.3. User Defined Functions

The user defines functions by creating function definition files. Each file defines a single function, and has the form:

DEF *name* AS *function*;

where *name* is a string and *function* is an IFP function, which is usually other functions combined by program forming operations. The definition may be freely formatted and occupy more than one line. For example, a function which doubles its input can be written:

```
DEF Double AS
   [id,id] | +;
```

(The meaning of the brackets and "|" is explained in the next section.)

3.3. Program Forming Operations

In conventional languages, functions are combined by storing intermediate results in variables. In IFP, functions are combined by "program forming operations" (PFO). Each component function is a *parameter* to the PFO. Like its component functions, each constructed function has a single input and output. Higher level functions can be built from the constructed functions in the same manner. The language APL[Ive62] provides a few PFO's (called *operators*), but their parameters are restricted to primitive functions.

3.3.1. Composition

IFP composition is identical to mathematical composition, except that it is written:

$$f \mid g$$

whereas it would be written mathematically as

$$g \circ f$$

Writing composition backwards follows from IFP's left to right syntax. The vertical bar is borrowed from UNIX, since composition of functions closely resembles piping between processes. Composition is defined by the equality:

$$x:f|g \equiv (x:f):g$$

Composition is associative, so parentheses are neither necessary nor allowed to indicate association in functions such as:

$$f \mid g \mid h$$

3.3.2. Construction

The construction of functions is written as a bracketed list of the functions. For example, the construction of functions f, g, and h is written as:

$$[f, g, h]$$

and defined by:

$$x:[f,g,h] \equiv <x:f,x:g,x:h>$$

3.3.3. Constant

The ***constant*** PFO creates a constant function. Constant functions always return the same result when applied to any object that is not "?". Constant functions are written as an object (the value to be returned) preceded by a pound sign "#". The defining equation is:

$$x : \#c \rightarrow \begin{cases} c \text{ if } x \neq ? \\ ? \text{ if } x = ? \end{cases}$$

3.3.4. Selector

The *selector* PFO creates selector functions, which behave as constant subscripts. Selector functions return the nth element of a sequence and are written as n, where n is a positive integer. There are also a corresponding set of select-from-right functions, written as $n\,r$. The defining equations are:

$$<x_1,x_2,\cdots> : n \equiv x_n$$

$$<\cdots,x_2,x_1> : n\,r \equiv x_n$$

3.3.5. Apply to Each

The *apply-to-each* PFO is a distributive form which applys a function to each element of a sequence. It is written as

EACH f END

The defining equation is:

$$<x_1,x_2,\cdots> : \text{EACH } f \text{ END} \equiv <x_1{:}f, x_2{:}f, \cdots>$$

Note that construction and apply-to-each are complementary. The former applies multiple functions to the same argument, the latter applies the same function to multiple arguments.

3.3.8. If-Then-Else

The *if-then-else* PFO allows conditional function application. It is written as

IF p THEN g ELSE h END

and the defining equation is:

$$x : \text{IF } p \text{ THEN } g \text{ ELSE } h \text{ END} \rightarrow \begin{cases} g(x) \text{ if } p(x)=\text{t} \\ h(x) \text{ if } p(x)=\text{f} \\ ? \quad \text{otherwise} \end{cases}$$

The level of nesting of conditional forms may be reduced by using ELSIF clauses. For example:

```
IF p1 THEN g1
ELSE
   IF p2 THEN g2
   ELSE
      IF p3 THEN g3
      ELSE h
      END
   END
END
```

can be rewritten as:

```
IF p1 THEN g1
ELSIF p2 THEN g2
ELSIF p3 THEN g3
ELSE h
END;
```

The "ELSE" part may not be omitted. Where the "ELSE" part is the identity function, it must be explicitly written as such.

3.3.7. Filter

The ***filter*** PFO filters through elements of a sequence that meet a criteria. It is written as:

```
FILTER p END
```

and is defined as:

```
EACH
   IF p THEN [id] ELSE [] END
END | cat
```

where p is the criteria expressed as a boolean function. For example, to filter a sequence for all pairs of equal elements:

```
<<a a> <c d> <y y> <r g>> : FILTER = END  →  <<a a> <y y>>
```

The filter PFO is an IFP extension to Backus' FP. Clearly it is not necessary since it could always be replaced by its definition, but it occurs sufficiently often to merit its inclusion as a PFO.

3.3.8. Right Insert

The *insert* PFO reduces a sequence. Its defining equations are:

$$<>\text{: INSERT } f \text{ END} \rightarrow ?$$

$$<x_1>\text{: INSERT } f \text{ END} \rightarrow x_1$$

$$<x_1, x_2, \cdots x_n>\text{: INSERT } f \text{ END} \rightarrow <x_1, <x_2, \cdots x_n>\text{: INSERT } f \text{ END } >\text{: } f$$

The name *insert* comes from picturing the PFO as inserting its parameter function between each element of a sequence, e.g.:

```
<1 2 3 4> : INSERT + END → 1+2+3+4
```

with right association.

Functions formed with *insert* are always undefined for empty sequences. This differs from Backus' FP, in which such functions returned the right identity element of the parameter function. In theory, this is a convenient feature (e.g. summing an empty sequence would yield 0), but it is impractical for the interpreter to know the identity element of user-defined functions. The number of cases where the interpreter could know the identity element are so few that we might as well define special functions for those cases, e.g:

```
DEF sum AS
    IF null THEN #0
    ELSE INSERT + END
    END;
```

Alternatively, we can append the identity element to the end of the sequence before inserting, e.g.:

```
DEF sum AS
   [id,#0] | apndr | INSERT + END;
```

Note that *insert* starts at the right of the sequence. Currently there is no "left insert", which would reduce a sequence starting from the left. We can get the same effect, however, with *insert* and the sequence reversal function:

```
reverse | INSERT reverse|f END
```

3.3.9. While

The *while* PFO is written as:

```
WHILE p DO f END;
```

and is defined by the recursive functional equality:

```
WHILE p DO f END ≡ IF p THEN
                       f | WHILE p DO f END
                   ELSE id
                   END
```

That is the *while* PFO applies the fewest *f*'s such that *x:f:f:f....:f:p* is true.

3.4. IFP Environment

In the good old days, card decks went in and printouts came out. IFP attempts to provide a more productive interface. The interpreter presents the user with an environment similar to UNIX or MS-DOS, depending upon which is the operating system of the host machine. The UNIX version is discussed here; the MS-DOS version is essentially identical except for command spellings.

When it is ready to accept a command, IFP prompts with:

```
ifp>
```

The user may edit a source file with any convenient editor. The edit command is the same as the UNIX command, for example if the editor is "vi", the user would respond to the "ifp>" prompt with:

```
vi foo
```

where *foo* is the name of the function to be edited.

To evaluate a function, the user enters the command "show" followed by an expression. For example, the user may enter:

```
show <3 4> : [+,-];
```

and the interpreter responds with:

```
<7 -1>
```

Programs rarely work right the first time. Therefore a language implementation should provide for debugging features. One problem of functional languages is that most debugging tools (such as traces and dumps) are side effects. Their practical value, however, far outweighs their blemishing presence. Perhaps truly functional debugging tools can be found in the future.

3.4.1. Back Tracing

One particularly unfortunate feature of FP's definition is that functions cannot print error messages, but instead return the value *bottom*, which represents an undefined value. For example, if the function application:

```
<1 2 3> : HorribleHairyFunction
```

returns "?", the user has no idea of which kind of error occurred. It could have been a division by zero, subscript out of bounds, or something else, but in all cases the same value is returned. For example, consider the inner product function application:

```
<<1 2 3> <4 5 w>> : Inner;
```

IFP indicates both the reason for the creation of "?", the offending argument, and propagation of the "?" result:

```
/math/arith/+: not a numeric pair
<3,w>
 | | |EXIT>  ? : /math/arith/+
 | |EXIT>  ? : EACH /math/arith/+ END
 |EXIT>  ? : /sys/trans|EACH .. END|/math/arith/sum
EXIT>  ? : /tmp/Inner
```

The functions are printed from their internal representation by an unparser, that is an elaborate pretty-printing routine. The dot pair ".." is used by the pretty-printer to abbreviate functions. The nesting level at which the dots occur may be set by the user with the command:

```
depth n
```

where *n* is the maximum level to be displayed. In the above example, the depth is 2; level 1 is the composition, level 2 is the *each* PFO, and level 3 is the parameter to *each*.

3.4.2. Tracing

IFP has two forms of tracing. The first form shows all applications and results inside a function. For example, if we have the function

```
DEF Mean AS              (* Mean of arithmetic sequence *)
   [sum,length] | %;
```

and then enter the commands

```
trace on Mean;
```

```
show <1 2 3 4> : Mean;
```

we get

```
ENTER> <1,2,3,4> : Mean
 |ENTER> <1,2,3,4> : [..,..]|"%"
 | |ENTER> <1,2,3,4> : [sum,length]
 | | |ENTER> <1,2,3,4> : sum
 | | |EXIT>  10 : sum
 | | |ENTER> <1,2,3,4> : length
 | | |EXIT>  4 : length
 | |EXIT>  <10,4> : [sum,length]
 | |ENTER> <10,4> : %
 | |EXIT>  2.5 : %
 |EXIT>  2.5 : [..,..]|%
EXIT>  2.5 : Mean
2.5
```

The second form of tracing lets us examine a specific intermediate result. This tracing is done with the special PFO "@". The function

@*string*

prints *string* followed by its argument. Otherwise "@" is an identity function. For example, if we want to see the argument to "%" in the *Mean* function, we can write:

```
DEF Mean AS
    [sum,length] | @"Before % in Mean" | %;
```

When this version of *Mean* is applied to <1 2 3 4>, we get the message:

```
Before % in Mean: <10 4>
```

CHAPTER 4.

Interpreter Design

A language interpreter is simply a program. Programs have two parts: data structures and algorithms. This chapter describes the IFP interpreter's data structures and algorithms. The interpreter is written in C[Ker78]; therefore examples of the interpreter's internal workings are also presented in C. The examples are simple enough that the reader need not be proficient with C.

4.1. Data Structures

Internally, IFP distinguishes eight types of objects. Each object is a discriminated union. The object types are:

```
BOTTOM
BOOLEAN
INT
FLOAT
LIST
STRING
NODE
CODE
```

Most of the type names are self descriptive. The NODE type is a compressed representation of a function pathname sequence. (See section 4.1.2.1.) The CODE type contains a pointer to machine code. It is internal to the interpreter and not directly available to the user. The representations of the other types are described in subsequent sections.

4.1.1. Scalar Types

The BOTTOM, BOOLEAN, INT, and FLOAT types all take a small fixed amount of storage. Each is stored in the machine's native format. The user can never distinguish the INT and FLOAT types, so the interpreter may change the representation at any time.

4.1.2. Strings

Strings are atoms in IFP, but require special treatment since they are variable-length entities. IFP strings are stored as linked lists of segments. Each segment contains approximately 12 characters. (The exact number is machine dependent.) The first record of the linked list contains the reference count for the string.

The string representation was chosen for the ability to allocate and deallocate strings quickly. Compaction of memory is never necessary. With 12 characters per segment and a 4 byte link field, we effectively use 75% of the memory available if internal fragmentation is not taken into account. On the average, half of the last segment for a string is empty, so the average internal fragmentation cost is 8 bytes per string. The strings also have a reference count. The intent was to allow conversions of call-by-value to call-by-reference, but none of current string operations use this feature.

Very few string primitives are implemented. Most string operations in IFP are done via *explode* and *implode*, which convert a string to a sequence of characters and vice versa. For example, to drop the first character from a string, we write:

```
explode | tl | implode
```

The string representations of single characters reside permanently in an array so that *explode* can operate quickly.

4.1.3. Sequences

IFP programs generate many sequences on the fly, so the sequence representation must allow quick allocation and deallocation of memory. Therefore IFP sequences are represented as linked lists. The lists are guaranteed to be acyclic by the definition of FP. That is, no sequence can ever contain itself as a sequence, so all sequences form trees.

```
typedef struct ListCell {
   Object Val;              /* Element of sequence     */
   struct ListCell *Next;   /* Pointer to next element */
   unsigned short LRef;     /* Reference count         */
} ListCell;
```

The one exception in which an indirect cycle does occur is in recursive function definitions. These cycles are broken, however, when the function name is deleted.

There were two choices for garbage collection: a mark/sweep algorithm or reference counts. Many of IFP's sequences are temporarily created by the structural functions. For the benchmark program in Chapter 6, Berkeley FP spends approximately 25% of its time garbage collecting with a mark/sweep algorithm. Reference counting was chosen for IFP, though it is not clear if it is an improvement.

Reference counting introduces the problem of count overflow. With most languages, the possible solutions are:

1. Make the reference count field big enough so overflow never occurs.

2. Implement a "sticky" reference count. Once the reference reaches its limit, it can not be decremented. A garbage collector must reclaim the storage.

IFP functions do not have side effects. Therefore the sharing of lists does not affect a program, and *not* sharing lists has no effect. Thus we have another solution to the reference count overflow problem:

3. Copy the data. Actually, the interpreter just copies the offending node and set its *Next* link to the *Next* link of the original node. The whole list does not have to be recopied, unless every list element's reference count has reached its limit.

The use of a single type of list-cell is somewhat space inefficient, since each cell must take up the worst-case space. For example, a boolean value occupies as much space as a double-precision floating-point number. The advantage of having the generic list-cell is that we can always replace its value without checking whether it would fit. The generic list-cell also reduces portability problems, since relative sizes of data types vary for different machines and compilers.

One way to shrink the list-cell would be to use references instead. For example, a list-cell holding a floating-point number could contain a pointer to the number instead of the number itself. On a typical 32-bit machine, this would reduce a 64-bit floating-point number to a 32-bit pointer, thus saving four bytes. Since a list-cell occupies approximately 16 bytes, this would yield a 25% saving of list-cell memory. This scheme would increase the interpreter's complexity and memory fragmentation, since a separate memory space would be allocated for floating-point numbers. The savings would be machine dependent also. On the CRAY numbers and pointers are the same size, in this case the reference scheme would take *extra* memory.

4.1.4. Functions

Backus FP represents functions as objects. In FP primitive functions are represented by atoms. For example, the atom *trans* would represent the transposition function. Functions created by PFO[1]

are represented as sequences: the first element is the PFO name and the rest of the elements are parameters to the PFO. The IFP interpreter uses the same scheme, except that function and PFO names are pathname sequences.

For example, the inner product function:

```
trans | EACH * END | INSERT + END
```

is represented as:

```
<
     <sys compose>
     <sys trans>
     <<sys each> <math arith *>>
     <<sys insertr> <math arith +>>
>
```

Table 1 shows the internal representations of PFO.

PFO	Representation
l l.	
#*c*	<<sys constant> #*c* >
#*?*	<<sys constant>>
n	<<sys select> *n* >
n r	<<sys select> −*n* >
$f_1 \mid f_2 \mid \cdots f_n$	<<sys compose>, $f_1, f_2, \cdots f_n$ >
$[f_1, f_2, \cdots f_n]$	<<sys construct>, $f_1, f_2, \cdots f_n$ >
EACH *f* END	<<sys each> *f* >
FILTER *p* END	<<sys filter> *p* >
INSERT *f* END	<<sys insertr> *f* >
IF *p* THEN *g* ELSE *h* END	<<sys if> *p g h* >
WHILE *p* DO *f* END	<<sys while> *p f* >

Table 1

ELSIF clauses are always expanded into equivalent nested IF-THEN-ELSE constructs. The representation of "#?" is a special case, because the representation <<sys constant> ?> is

[1]Program forming operations (PFO) are defined in section 3.3.

equivalent to "?" by the bottom preserving property.

When evaluating a function application, the interpreter must look up the code corresponding to the function or PFO pathname sequence. To speed up the search, pathname sequences are converted to an alternative representation (type NODE) that is a direct pointer to the function.

4.1.5. Environment

Functions are stored as UNIX files. There is simply a UNIX file tree which corresponds to the user's function tree. This is effective as an interpretive environment. The user's favorite editor can be used to edit a function file. When a change is made, only the file corresponding to the altered function must be read and parsed again, thus speeding up incremental modification.

The interpreter loads function definitions on a demand basis. Currently there is no memory release mechanism in the interpreter. Once a function is loaded, it remains resident until it is modified or the interpreter exits.

Furthermore, the use of UNIX files for function definitions allows the use of UNIX utilities. For example, the user can list directories with the *ls*, list functions with *more*, and search for patterns with *grep*. Not only does this save the implementor time, but creates a familiar user interface.

The only problem is that the interpreter must recognize some of the UNIX commands. Consider the *rm* command. It will delete a file containing a function. If the function is already resident in the interpreter, the internal representation must be removed. Therefore the interpreter must recognize *rm* and remove the function from its internal storage. Currently only the "vi" and "rm" commands are recognized. In both cases, the internal copy of the function is removed. (The new version of the edited function will be loaded on demand.) The "mv" and "cd" commands should also be recognized as special, but are currently not. (Since IFP forks off UNIX commands, "cd" only changes the child's environment, not that of the interpreter!)

4.2. Algorithms

IFP has a single evaluation operation *apply*, which applies a function to an object to yield another object. In IFP, the application of function f to object x is written as:

$$x:f$$

Internally, the interpreter contains the C function:

```
void Apply (InOut,F)
   ObjectPtr InOut,F;
```

The value of **InOut*[2] is replaced by the result of applying function **F* to object **InOut*. There are three types of functions:

Primitive Functions

The function is defined by machine code in the interpreter. All primitive functions have the same format:

```
void F_foo (InOut)
   ObjectPtr InOut;
   {
      ... /* Code for primitive function foo */
   }
```

In C, **p* denotes the value pointed to by *p*, and *&X* denotes a pointer to variable *X*. C uses call-by-v
clusively, call-by-reference is done by passing a pointer to the argument

To evaluate its application, the machine code is executed with InOut pointing to the input object. The input object is replaced by the result of the application.

User defined functions

The function is defined by the user. To evaluate its application, the function's definition is applied to the argument.

Compound functions

The function is the result of a PFO. The PFO is defined by machine code. To evaluate the function, the machine code is executed with the PFO's parameters as additional arguments. The machine code for a PFO is essentially a control structure which selectively applies the parameter functions to the input. For example, the code for evaluating function composition is:

```
Compose (InOut,Funs)
   ObjectPtr InOut;
   ListPtr Funs;
   {
      while (Funs != NULL) {
         Apply (InOut, &Funs->Val);
         Funs = Funs->Next;
      }
   }
```

where *InOut* is the input to the composition PFO and *Funs* is the list of functions to be composed. *Inout* serves as an accumulator; the code simply traverses the function list and applies each function to the accumulator.

Note that *Apply* does not recursively call itself for primitive function evaluation. If we wanted to convert tail-recursions to iterations, it is only the PFO's we would have to reimplement; the primitive functions would remain unchanged.

The theoretical order of evaluation in FP and the actual order in the interpreter are quite different. In principle, all PFO's combine their parameters to create a new function. The result function is then applied to the object. This is the usual order in mathematics. For example, to evaluate:

$$\left[D_x f(x)\right](2)$$

we first evaluate $D_x f(x)$ and then apply f' to the argument 2. The actual evaluation is a top-down recursive procedure. For example, to evaluate:

```
<3 5> : EACH [id,#1]|+ END;
```

The following applications occur:

```
<3,7> : EACH [id,#1]|+ END
   3 : [id,#1]|+
      3 : [id,#1]
         3 : id
         3 : #1
      <3,1> : +
   7 : [id,#1]|+
      7 : [id,#1]
         7 : id
         7 : #1
      <7,1> : +
```

Essentially, *Apply* is a threaded code interpreter similar to FORTH[Loe81]. The difference between IFP and FORTH is in the form of the code and data. FORTH interprets linear code which specifies transformations of a stack. IFP interprets list-structured code which specifies transformations of a list.

4.2.1. Conversion of Call-by-Value to Call-by-Reference

The reduction evaluation of FP expressions requires call-by-value arguments. Implementing call-by-value by always copying arguments would be expensive in both time and space, since FP arguments are often complicated structures. Therefore IFP uses call-by-reference internally. Since functions cannot modify their arguments, call-by-reference is indistinguishable from call-by-value. If a function (such as *reverse*) needs to return a modified version of its argument, it makes a local copy first. This is analogous to the copy-on-write scheme for operating systems in which a process image is not copied after a fork until the parent or child process needs to modify the image.

The internal use of call-by-reference does not eliminate all unnecessary copying. Consider a case of copy-on-write in which the child process wants to modify part of the process image, but a part which is no longer used by the parent process. An example of this in IFP is the expression:

```
X : trans | EACH reverse END
```

which rotates matrix X 90° clockwise. In a simple interpreter, *trans* would make a local copy of X and transpose the local copy. The *each* PFO would then pass each row of the transposed matrix to *reverse*. On each application of *reverse* to a row, it would first make a local copy of the row, and then reverse the row. The copy operation is redundant, however, since the row handed to *reverse* was a local copy already. That is, it doesn't matter if *reverse* alters its argument in this case, since it has sole possession of the argument. The general rule is: a function may directly modify the section of a list for which the reference counts are all unity. Statistics for actual IFP programs show that approximately 20–50% of all list cells are "recycled" this way rather than created from scratch.

The copy avoidance is simple to implement, it is encapsulated in two procedures:

```
void CopyTop (A)
   ListPtr *A;

void Copy2Top (A)
   ListPtr *A;
```

The former effectively generates a copy of list *A, but doesn't bother to copy the prefix of the list with unit reference counts. The latter effectively copies the top two levels of *A. Deeper copying is not required by any of the primitive functions.

4.2.2. Pointer Rotation

Since garbage collection is done with reference counts, the interpreter must be coded very carefully. If a reference count is too low, an object will be prematurely snatched by the garbage collector. If a reference count is too high, an unreferenced object will never be collected. Of particular hazard are pointers local to a procedure. They are allocated upon procedure entry, and more importantly, automatically disappear upon procedure exit.

The simplest way to maintain reference counts is to have functions which do the book keeping. For example, we could initialize all pointers to NULL, and then do all pointer assignments via the function:

```
void RepLPtr (A,B)
   ListPtr *A, B;
```

which would replace pointer *A by pointer B and adjust reference counts appropriately. The overhead, however, is considerable. Consider the sequence of pointer assignments from a list reverse routine:

```
Q = T = NULL;
while (R!=NULL) {
   T = R;
   R = R->Next;
   R->Next = Q;
   Q = T;
}
R = Q;
Q = T = NULL;
```

Except for the NULL assignments, all the assignments would have to be done via the

RepLPtr operation (if we were strict about using our pointer copying procedure). Note, however, that the reference counts are actually unchanged once the reversal is done.

In fact, many pointer manipulations conserve reference counts. A frequently occurring manipulation is *pointer rotation*[Suz80]. A pointer rotation cyclicly permutes a set of pointers. For example, a three-way rotation procedure is:

```
void Rot3 (A,B,C)
   ListPtr *A,*B,*C;
   {
      char *P;
      P = *A; *A = *B; *B = *C; *C = P;
   }
```

A pointer rotation does not modify reference counts. By using rotations as a primitive pointer operation, most of the reference count modifications can be avoided. For example, the list reversal procedure can be rewritten as:

```
Q = NULL;
while (R!=NULL) Rot3 (&R, &R->Next, &Q);
Rot2 (&R, &Q);
```

(The Rot2 procedure swaps two pointers.) All the reference counting overhead disappears. Higher order rotations are also useful, the interpreter even does five-way rotations within the distribute-left function. (Though the five-way rotation could be replaced by two 3-way rotations.)

4.2.3. Vectorising List Manipulation

For dealing with linked structures the most common memory allocation primitive for linked structures allocates a single record. An example is the Pascal *new* procedure. IFP, however, typically does not work a word at a time. Therefore a faster and more convenient primitive is implemented:

```
void NewList (A,N)
   ListCell **A;
   long N;
```

NewList points *A to a fresh list of N cells all set to "?"; the last cell of the list points to the old value of *A. That is insert N new cells at the head of list pointed to by *A. To simply get a list of N cells, *A's value is first set to NULL. The NewList operation sounds unnecessarily complex for a primitive, but actually has the important property that it preserves reference counts.

4.2.4. Expression Cache

Since FP expressions are referentially transparent, evaluating a given expression yields the same result every time. If an expression occurs twice, the interpreter needs only evaluate the expression once and remember the result. To remember previous results the IFP interpreter has an *expression cache*[Kel86]. The cache associates an expression (*input:function*) with an *output* value.

The cache is implemented as a hash table. Before an expression is evaluated, it is first mapped to a hash table index. If the corresponding table entry is full, then the entry's expression is compared with the expression to be evaluated. If the two are equal, then the associated output value is taken from the cache. Otherwise the expression must be evaluated and the <*expression,output*> association is stored in the cache for future reference. Collisions are resolved by evicting the previous entry from the cache.

The cache lookup speed is limited by the time it takes to hash and compare structures. Since both these operations take $O(n)$ time[3] the cache lookup operation takes $O(n)$ time.

[3] O denotes a lower bound; Ω denotes an upper bound.

Most of the current primitives take at least $\Omega(n)$ time.[4] Since the expressions with primitive functions can almost always be computed more quickly than accessing the cache, they are always evaluated, and never looked up in the cache. The interpreter looks in the cache just for user-defined function applications. Experiments with the interpreter support the above reasoning: caching of primitive functions slowed the interpreter down by 27-76%. A third possibility would be to cache expressions involving PFO. This was not implemented, though the results would probably be the same as for primitives. The real gains from a cache occur with user-defined recursive functions as described below.

The interpreter may be compiled with or without the cache mechanism. For most programs, the extra lookup operation slows down the interpreter, but for certain combinatorial programs the cache can change the program's asymptotic time. One such program computes the nth Fibonacci number:

```
DEF Fib AS
   IF [id,#2] | < THEN id
   ELSE
      sub1 | [Fib,sub1|Fib] | +
   END;
```

(The *sub1* function returns one less than its argument.) Without the cache, the program takes time $O(\phi^n)$, where ϕ is $\frac{1+\sqrt{5}}{2} = 1.618....$ The reason is that computations form a tree as shown in figure 3. The tree shows that to compute $f(n)$, the program must first compute $f(n-1)$ and $f(n-2)$ The dotted-lines indicate redundant parts of the tree. With the cache, the interpreter descends the left side of the tree. As the interpreter ascends the left side of the tree, the result of each right subtree (except $f(0)$) has already been computed and loaded into

[4]The two exceptions are *iota* and *repeat*, which are defined as k:iota $\rightarrow$ $<1,2,3, \cdots k>$ and $<x\ k>$:repeat $\rightarrow$ $<k$ repetitions of $x>$. Clearly their execution time is linear in k and independent of the structural size of their input.

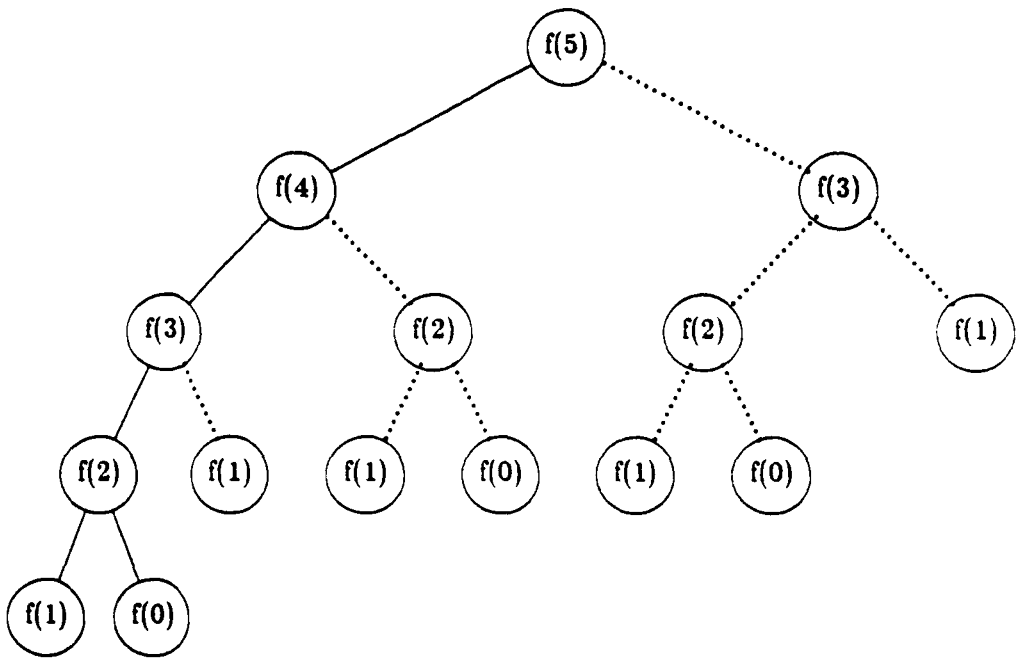

Figure 3

the cache. The cache dynamically merges the tree's branches so that the execution time is $O(n)$.

An interesting phenomena is that the speedup may exceed the cache hit rate. The cache hit rate is defined as:

$$\frac{\text{number of results found in cache}}{\text{number of results looked for in cache}}$$

In the Fibonacci number case, the left subtree must always be computed, and the right subtree (except $f(0)$) is always found in the cache. Therefore the the hit rate asymptotically approaches 50%. The speedup is defined as:

$$\frac{\text{execution time without cache}}{\text{execution time with cache}}$$

Without the cache, all the nodes in the tree must be evaluated. The recurrence relation for the number of nodes t_n in the tree is:

$$t_0=1$$
$$t_1=1$$
$$t_n=t_{n-1}+t_{n-2}+1$$

From this recurrence it follows that:

$$t_n = O\left(\phi^n\right)$$

With the cache, the n left-most nodes and $f(0)$ are computed; The $n-1$ right subtrees of these nodes are looked up in the cache. The total computation time is $\Omega(n)$. Thus the speedup is:

$$O\left(\frac{\phi^n}{n}\right)$$

which asymptotically approaches ∞ even though the hit rate approaches only 50%.

The speedup argument assumes that no collisions occur in the cache's hash table. The current hash table contains 1024 entries. The effect of collisions is difficult to assess. Cache access are far from random; they are dependent upon the function being computed. For the Fibonacci function only the two most recently evaluated expressions contribute to the next expression, so no significant collisions occur. If these fortuitous circumstances did not exist, then the analysis is much more complex. A sketch of one possible analysis follows. Suppose that:

$$p = \text{cache miss rate} = 1 - \text{cache hit rate}$$

$$c = \frac{\text{time for addition}}{\text{time for cache lookup}}$$

The average execution time t_n for evaluating $f(n)$ is the weighted average of two possibilities: either $f(n)$ is in the cache, or $f(n)$ must be computed from $f(n-1)$ and $f(n-2)$. The recurrence is:

$$t_0 = (1-p)+pc$$
$$t_1 = (1-p)+pc$$
$$t_n = (1-p) + p\left[t_{n-1}+t_{n-2}+c\right]$$

The roots of the homogeneous characteristic equation are:

$$r_1,r_2 = \frac{p \pm \sqrt{p^2+4p}}{2}$$

Taking r_1 as the larger root, the asymptotic approximation of t_n takes the form:

$$t_n = O\left(r_1{}^n\right)$$

for $r_1>1$. In the perfect situation, the cache miss-rate approaches 0.5. Suppose that collisions cause the cache miss-rate to rise to 0.6. Then the average execution time is:

$$t_n = O\left(1.130...^n\right)$$

Though not linear, the time is still asymptotically better than the time for uncached evaluation, $O\left(1.618...^n\right)$. This analysis, however, assumes constant and independent cache-miss probabilities, which is known, as mentioned earlier, to be a false assumption for the Fibonacci function. Therefore applying similar logic to other combinatorial programs is an estimate only and may not be realistic.

CHAPTER 5.

IFP Example Programs

This chapter presents some IFP programs. The programs are selected to demonstrate the power and elegance of functional programming and the IFP notation.

5.1. Tangent

The tangent of an angle is the quotient of its sine and cosine. An IFP tangent function is shown in listing 1. Comments are delimited by "(*" and "*)". Note that the independence of the sine and cosine calculations is explicit in IFP. The sine and cosine can be calculated in any order, or in parallel. Furthermore, the program expresses the computation without intermediate variables.

```
(*
 * Tangent
 *
 * Compute the tangent of an angle expressed in radians.
 *
 * E.g. 0.7854 : Tangent -> 1
 *)

DEF Tangent AS [sin,cos] ¦ %;
```

Listing 1

5.2. Greatest Common Factor

The greatest common factor of two positive integers can be defined recursively:

$$\mathrm{gcf}\,(a,b) = \begin{cases} a & \text{if } a = b \\ \mathrm{gcf}\,(a-b,b) & \text{if } a > b \\ \mathrm{gcf}\,(b-a,a) & \text{if } a < b \end{cases}$$

An IFP translation of these equations is show in listing 2. Since program is tail recursive, it may be transformed with the *while* PFO into the program shown below.

```
DEF gcf AS
   WHILE ~= DO
      IF > THEN id ELSE reverse END |
      [-,2]
   END | 1;
```

(The function ~= test for inequality of two objects.)

```
(*
 * gcf
 *
 * Compute greatest common factor of a numeric pair.
 *
 * Example:
 *                    <144 128> : gcf -> 16
 *)

DEF gcf AS
   IF = THEN 1
   ELSE
      IF > THEN id ELSE reverse END |
      [-,2] | gcf
   END;
```

Listing 2

5.3. PowerSet

The powerset of a set is the set of all subsets. The powerset may be recursively defined as:

$$\text{PowerSet}(\{\}) \equiv \{\{\}\}$$

$$\text{PowerSet}(\{x\} \bigcup S) \equiv \bigcup_{T \epsilon \text{PowerSet}(S)} \left\{ T, \{x\} \bigcup T \right\}$$

These equations translate almost directly into the FP program in listing 3.

```
(*
 * PowerSet
 *
 * The PowerSet function generates all subsets of a given set.
 * Sets are represented as sequences of distinct elements.
 *
 * Examples:
 *
 *        <> : PowerSet -> <<>>
 *
 *        <a b c> : PowerSet -> <<a,b,c>,<a,b>,<a,c>,<a>,<b,c>,<b>,<c>,<>>
 *)

DEF PowerSet AS
    IF null THEN [id]
    ELSE
        [1, tl ¦ PowerSet] ¦
        [
            distl ¦ EACH apndl END,
             2
        ] ¦ cat
    END;
```

Listing 3

5.4. QuickSort

Hoare's quicksort algorithm partitions a sequence by comparing each element against a chosen element. Two subsequences are formed: one subsequence contains all elements less than the chosen element, and the other subsequence contains the remaining elements. The two subsequences are then sorted recursively and the sorted subsequences are catenated. Of course the case of two or fewer elements in the input sequence is trivial: the algorithm simply returns the input sequence.

To simplify the algorithm's symmetry, the IFP program in listing 4 partitions the sequence in subsequences of elements less-than, equal-to, and greater-than the chosen element. The IFP program expresses the quicksort idea (partitioning on a key) without the complex memory shuffling required by word-at-a-time languages such as Pascal.

```
(*
 * QuickSort
 *
 * Sorts a sequence of numbers into ascending order using quicksort algorithm
 *
 * E.g. <3 2 6 4 5 8 0> : QuickSort -> <0 2 3 4 5 6 8>
 *)

DEF QuickSort AS
   IF [length,#2] ¦ < THEN id  (* Check for trivial case *)
   ELSE
      [id,1] ¦ distr ¦            (* Distribute partition key over sequence *)
      [
         FILTER < END ¦ EACH 1 END ¦ QuickSort,  (* Sort lower partition *)
         FILTER = END ¦ EACH 1 END,
         FILTER > END ¦ EACH 1 END ¦ QuickSort   (* Sort upper partition *)
      ] ¦ cat
   END;
```

Listing 4

CHAPTER 6.

Performance

6.1. Speed

This section compares the execution speed of the IFP interpreter with the Berkeley FP interpreter and conventional von-Neumann languages.

6.1.1. Illinois FP vs. Berkeley FP

The LU decomposition program in appendix C was run on both interpreters. Two times were computed for each interpreter: load and execution. The load time includes reading source files and parsing, which is overhead independent of the number of executions of the program. The execution time is the additional time required for each execution of the program.

The load and execution times were computed from a linear regression. The interpreters were started and the benchmark program run n times before exiting the interpreter. A linear regression on the total time vs. n yield the load and execution times as the Y-intercept and slope respectively.

The benchmark results are shown in table 2. All times are in seconds. The Berkeley FP does not interpret FP directly, but translates it into LISP. The LISP may be compiled with the Liszt LISP compiler[Fod83]. The resulting code still ran much more slowly than the IFP interpreted code as shown in table 3.

Interpreter	Load (sec)	Execute (sec)
Berkeley FP 4.2	68	119
Illinois FP 0.4	0.90	3.18
ratio	75.6	37.4

Table 2

Code	Compile Time (sec)	Load (sec)	Execute (sec)
Berkeley FP 4.2 (compiled)	138.5	4.7	90
Illinois FP 0.4	0	0.90	3.18
ratio	-	5.22	28.3

Table 3

Comparing relative code sizes of the FP interpreters is difficult. Table 4 lists the approximate source and object sizes of the interpreters. The executable file for Berkeley FP is actually 770k, but the Berkeley FP interpreter is built on top of FRANZ Lisp, $^{\text{Fod83}}$ and thus most of the interpreter (~639k) is actually the LISP interpreter. Of the remaining 131k, approximately 16% collects statistics. Thus the other 84% (110k) gives about the same functionality as IFP. The Illinois FP executable file is 42k. For the source line counts, each source was stripped of comments. In the case of FRANZ lisp, the statistics package source was omitted.

Interpreter	Source Language	Source Lines	Object Size
Berkeley	LISP	1900	110k
Illinois	C	4700	42k

Table 4

6.1.2. Illinois FP vs. von-Neumann Languages

The IFP interpreter was also compared against interpreted BASIC and compiled Pascal. The LU decomposition benchmark was rewritten in BASIC and Pascal. The IFP and BASIC versions in appendix C were run on an IBM PC/AT with the MS-DOS operating system. Both the IFP and BASIC programs were loaded into memory before execution. The resulting times are shown in table 5.

Interpreter	Execution Time
BASIC	9.6 sec
IFP	11.0 sec
ratio	0.87

Table 5

That the IFP version runs only 13% slower than the BASIC version is remarkable. The BASIC code has several significant advantages. The BASIC interpreter is presumably written in assembly language. the BASIC program takes advantage of it's von-Neumann model by computing the LU decomposition in place. Furthermore, the BASIC version avoids computing or using the zeros implicit in L and U, thus saving operations. The IFP interpreter is written in C. The IFP program (see appendix C) does not compute the LU decomposition in place, computes the implicit zeros in L and U, and computes the subexpression *Aik* twice as often as its BASIC counterpart, since *L* and *U* both call *Aik*. Evidently the redundant computations are not a serious impediment.

The benchmark of IFP against BASIC shows that IFP is not the terribly inefficient language it might appear to be. On first inspection, the IFP structural functions *distl, distr,* and *trans* seems inefficient compared to the use of array subscripts in BASIC. In an interpreter, however, the FP structural functions have the advantage that they need only be interpreted once per application. The corresponding BASIC subscripts must be interpreted each time through the loop. For example, the IFP code:

```
[A,B] | trans | EACH * END
```

would be correspond to the BASIC code:

```
100 FOR J=1 TO N
110    C(J) = A(J) * B(J)
120 NEXT J
```

Each time through the loop each variable reference and subscript must be interpreted. Someone once noted "software slows down hardware." Here we find "variables slow down software!" This problem also occurs in compiled code, in which the hardware is interpreting an instruction stream. Vector machines such as the CRAY X-MP[1] essentially have instructions which combine a *distl, distr,* or *trans* with a subsequent *each* PFO.

Of course a compiler can remove the interpretive overhead. The times for the LU decomposition on a VAX for Pascal and IFP are shown in table 6. The Pascal program was compiled by the "pc" compiler with the optimizer turned on.

Language	Method	Compile	Load	Execute
Illinois FP	interpreter	–	0.90	3.18
Pascal	compiler	4.2	0.11	0.12
ratio		–	8.2	26.5

Table 6

6.2. Portability

The choice of C for writing the interpreter was quite beneficial. Not only is the interpreter fast, but it is portable to many different machines. Table 7 lists machines to which

CRAY X-MP is a trademark of CRAY RESEARCH, INC.

IFP has been ported. In most cases, only a few machine-dependent global constants (e.g word size) must be changed to port the interpreter.

Machine	Operating System
CRAY X-MP	CTSS
Pyramid 90x	4.2 BSD UNIX
IBM PC/RT	4.2 BSD UNIX
VAX 11/780	4.2 BSD UNIX
IBM S9000	XENIX
IBM PC/AT	XENIX
IBM PC	MS-DOS

Table 7

CHAPTER 7.

Conclusion

The IFP interpreter meets its original specification. The interpreter provides a simple environment for writing, debugging, and executing functional programs.

The choice of C for writing the interpreter was quite beneficial. Not only is the interpreter an order of magnitude faster than its LISP counterpart, but it is portable to many target machines.

IFP needs much more sophisticated data types. Currently, IFP could be described as a functional ALGOL–60. In particular, a record type and the corequisite accessing functions are needed. Taken further, the interpreter could include data–encapsulation features, which would allow for functional object–oriented programming.

APPENDIX A.

IFP Grammar

Character Set

IFP uses the ASCII character set. Upper and lower case letters are distinct.

Tokens

IFP's scanner is context sensitive. The context is determined by the parser. Tokens are the longest sequence of characters not containing a delimiter. Atoms are delimited by

space , < > ¦ [ ] ( ) ; : *tab* *newline*

and function names are delimited by:

space , [ ] ( ) ¦ ; : / *tab* *newline*

Comments are delimited by "(*" and "*)" as in Pascal, and are lexically equivalent to spaces. The delimiters for atoms and functions differ so that the comparison functions can be written "<", ">", ">=", and "<="; angle brackets within objects delimit sequences.

Strings may be in single or double quotes. Strings not quoted must not contain atom delimiters. Strings which look like other type atoms must also be quoted. That is the strings "t" and "f" must be quoted to distinguish them from boolean atoms; strings of digits must be quoted to distinguish them from numeric atoms.

IFP reserved words are always in upper case. The reserved words are:

AS DEF DO EACH ELSE ELSIF

IF INSERT FILTER THEN WHILE

Productions

Table 8 shows the EBNF ^{Wir77} production rules for IFP definitions. The *Representation* production simply allows the user to invoke the IFP parser to create object representing a function. The object created by *Representation* is the internal form of the function as described in section 4.1.2.1. For example, writing

```
#(EACH reverse END)
```

is the same as writing:

```
#<<sys each> <sys reverse>>
```

Def →	'DEF' String 'AS' Comp ';'
Comp →	Function ¦ '¦' Function ¦
Function →	Conditional ¦ Constant ¦ Construction ¦ Debug ¦ Each ¦ Filter ¦ Insert ¦ Pathname ¦ Select ¦ While
Conditional →	'IF' Comp 'THEN' Comp ¦ 'ELSIF' Comp 'THEN' Comp ¦ 'ELSE' Comp 'END'
While →	'WHILE' Comp 'DO' Comp 'END'
Select →	UnsignedInt [r]
Insert →	'INSERT' Comp 'END'
Each →	'EACH' Comp 'END'
Filter →	'FILTER' Comp 'END'
Debug →	'@' Object
Constant →	'#' Object
Construction →	'[' [Comp {',' Comp¦] ']'
Pathname →	['/'] String ¦'/' String}
Object →	Bottom ¦ Atom ¦ Sequence ¦ Representation
Representation →	'(' Comp ')'
Sequence →	'<' [Atom {',' Atom ¦] '>'
Bottom →	'?'
Atom →	Number ¦ String ¦ Boolean
Number →	Integer ['.' ¦digit¦ ['e' Integer]]
Integer →	['+'¦'-'] UnsignedInt
UnsignedInt →	Digit {Digit}
Digit →	0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9
Boolean →	't' ¦ 'f'
String →	' { character } ' ¦ " ¦ character ¦ " ¦ { character }

Table 8

Table 9 shows the productions for IFP import files. A typical import file is shown in listing 5. Imported function names must not conflict with other functions either local or imported to the module.

ImportFile →	{ Import }
Import →	'FROM' Pathname 'IMPORT' String { ',' String } ';'

Table 9

```
FROM /sys IMPORT
   apndl, apndr, cat, distl, distr, explode, id, implode,
   iota, length, pick, repeat, reverse, tl, tlr, trans;

FROM /math/arith IMPORT
   +, -, *, %, add1, arcsin, arccos, arctan,
   cos, div, exp, ln, mod, min, minus, max,
   power, sin, sum, sqrt, sub1, tan;

FROM /math/logic IMPORT
   =, ~=, >, <, >=, <=, ~, and, all, any, member, null, or;
```

Listing 5

Table 10 shows the productions for IFP interpreter commands. for the UNIX version of IFP; the MS–DOS version is quite similar. The variable *Editor* is the name of the user's editor. *UnixCommand* may be any UNIX command on the host system.

command ->	'show' Object ':' Comp |
	'trace' ('on' | 'off') Pathname { ',' Pathname } |
	'depth' UnsignedInt |
	'rm' Pathname |
	Editor Pathname |
	UnixCommand |

Table 10

APPENDIX B.

Differences between IFP and Backus' FP

Domain

Backus' FP has two types of atom, the string and the empty sequence. IFP atoms do not include the empty sequence. IFP include numbers and truth values as atoms distinct from strings.

Functions

There are many new primitives.

Functional Forms

Backus' FP defines the INSERT form for empty sequences as returning u_f, the right identity element of f. IFP defines INSERT as returning "?" for empty sequences.

IFP has a new functional form, FILTER, which is described in section 3.3.7.

Syntax

The IFP syntax is designed to facilitate indentation and comments. All functional forms bracket their parameters, so no parentheses are necessary to indicate association. Table 11 shows the syntactic differences between Backus' FP and IFP. Backus' functions occupy a flat name space. IFP functions are arranged in a tree structure and referenced by pathnames, which are lexically scoped.

Backus	IFP
$C \circ B \circ A$	A ¦ B ¦ C
$p \rightarrow f; g$	IF p THEN f ELSE g END
$p \rightarrow f;\ q \rightarrow g;\ h$	IF p THEN f ELSIF q THEN g ELSE h END
αf	EACH f END
/f	INSERT f END
(while p f)	WHILE p DO f END
(<u>bu</u> f x)	[id,#x] ¦ *f*
$\bar{f}$	#*f*
Def f ≡ x	DEF f AS x;
ϕ	< >
ppd	?

Table 11

APPENDIX C.

Benchmark Program

Listing 4 is the program used to compare execution speeds of the Berkeley and Illinois FP interpreters. The program computes the LU decomposition of a matrix, and then compares the product of LU with the original matrix. The difference should be 0, though due to rounding errors it is approximately 10^{-28}.

```
(*
 * Compute LU decomposition of matrix A, then take sum-square-error
 * between LU and A.
 *)
DEF BenchMark AS
   A | [[L,U] | MatMul,id] | MatSub |
   cat | EACH [id,id]|* END | INSERT + END;

(* L part of LU decomposition of matrix *)
DEF L AS
   IF Singleton THEN #<<1.0>>
   ELSE
      [
         L11,
         A1k | [EACH #0 END,L] | apndl
      ] | ApndlCol
   END;

(* U part of LU decomposition of matrix *)
DEF U AS
   IF Singleton THEN id
   ELSE
      [
         U1k,
         A1k | [EACH #0 END,U] | ApndlCol
      ] | apndl
   END;
```

```
(* Tail of matrix after gaussian elimination on 1|1 *)
DEF A1k AS
   [
      MatTail,
      [L11|tl,U1k|tl]|Outer
   ] | MatSub;

(* First column of L part *)
DEF L11 AS [EACH 1 END,1|1] | distr | EACH % END;

(* First row of U part *)
DEF U1k AS 1;

(* Append column (1) to left side of matrix (2) *)
DEF ApndlCol AS [1,2|trans] | apndl | trans;

(* Inner product *)
DEF Inner AS
   trans | EACH * END |
   IF null THEN #0
   ELSE INSERT + END
   END;

(* Matrix multiplication *)
DEF MatMul AS
   [1,2|trans] |
   distr |
   EACH distl |
      EACH Inner END
   END;

(* Matrix subtraction *)
DEF MatSub AS MatCat | EACH EACH - END END;

(* Converts pair of matrices to matrix of pairs *)
DEF MatCat AS trans | EACH trans END;

(* Deletes first row and column of matrix *)
DEF MatTail AS tl | EACH tl END;

(* Outer product of two vectors *)
DEF Outer AS Cart | EACH EACH * END END;

(* Cartesian product of two vectors *)
DEF Cart AS distr | EACH distl END;
```

```
(* Check if square matrix is a singleton *)
DEF Singleton AS [length,#1]|=;

(* Input Matrix *)
DEF A AS
   #<
       < 2.3  4.7 -2.7  5.7  7.4 2.1 12.7  1.1 32.1  4.5  1.1  8.3>
       < 1.7 -1.7  5.2  3.2  1.2 3.5  2.4  2.9  1.9  1.7 -4.5 -9.9>
       < 6.1  3.4  1.2 10.6  2.9 1.7  1.1 -0.3  1.2  3.2  1.6  1.3>
       <23.3 -9.7  2.4  5.2  7.6 1.1 86.2  1.7  3.2  9.7  1.2 87.1>
       < 1.2  3.4  4.5  6.7  9.8 0.1  2.1  5.7 -9.1 -5.2  0.2  1.7>
       <12.3  1.2  8.7 12.3 -4.7 -.1  3.2  2.1  4.3  1.8  1.9  2.3>
       < 5.7  4.7 -2.8  5.7  7.4 2.1 12.7  1.1 32.1  4.5  1.1  8.3>
       < 1.7 -6.7  5.6  7.4  1.2 3.5  2.7  2.8  1.9  1.7 -4.5 -9.9>
       < 3.1 -3.4 -9.2 10.6  8.9 1.7 -1.1 -0.3  3.2  3.2  1.6  1.3>
       <13.3 -9.7  5.2  7.6 1.1 86.2  1.3  3.2  9.7  1.2 87.1 -9.2>
       < 1.2  3.4 -4.5 -6.7  9.8 0.1 -2.1  5.8 -9.1 -5.2  0.2  1.7>
       <12.3  1.2 -8.7 12.3 -4.7 -.1 -3.2  1.8  1.9  2.3  3.1  4.3>
   >;
```

Listing 4

Listing 5 shows a BASIC version of the LU decomposition. It calculates the LU decomposition of matrix A in place, i.e. after the decomposition L is in the lower triangle of A and U is in the upper triangle of A. The diagonal of 1's in L is implicit. Since IFP computes in double precision, the BASIC program also computes in double precision.

```
100 REM LU DECOMPOSITION BENCHMARK
110 DEFINT I,J,K,N
120 DEFDBL A,D,S
130 LET N=12
140 DIM A(N,N),A1(N,N),A2(N,N)
150 REM READ MATRIX A (AND SAVE IN A1 FOR LATER USE)
160 FOR I=1 TO N
170    FOR J=1 TO N
180       A(I,J)=RND
190       A1(I,J) = A(I,J)
200    NEXT J
```

```
210 NEXT I
220 REM COMPUTE LU DECOMPOSITION IN PLACE
230 FOR J=1 TO N
240    FOR I=J+1 TO N
250       A(I,J) = A(I,J) / A(J,J)
260       FOR K=J+1 TO N
270          A(I,K) = A(I,K) - A(I,J) * A(J,K)
280       NEXT K
290    NEXT I
300 NEXT J
310 REM MULTIPLY L AND U, PUT PRODUCT IN A2
320 FOR I=1 TO N
330    FOR K=1 TO N
340       S=0
350       IF I > K THEN M=K ELSE M = I
360       FOR J=1 TO M
370          IF I=J THEN S = S+A(J,K) ELSE S=S+A(I,J)*A(J,K)
380       NEXT J
390       A2(I,K)=S
400   NEXT K
410 NEXT I
420 REM COMPUTE SUM-SQUARE ERROR IN S
430 S=0
440 FOR I=1 TO N
450    FOR K=1 TO N
460       D = A1(I,K) - A2(I,K)
470       S = S + D * D
480    NEXT K
490 NEXT I
500 PRINT S
510 END
520 DATA  2.3  4.7 -2.7  5.7  7.4 2.1 12.7  1.1 32.1  4.5  1.1  8.3
530 DATA  1.7 -1.7  5.2  3.2  1.2 3.5  2.4  2.9  1.9  1.7 -4.5 -9.9
540 DATA  6.1  3.4  1.2 10.6  2.9 1.7  1.1 -0.3  1.2  3.2  1.6  1.3
550 DATA 23.3 -9.7  2.4  5.2  7.6 1.1 86.2  1.7  3.2  9.7  1.2 87.1
560 DATA  1.2  3.4  4.5  6.7  9.8 0.1  2.1  5.7 -9.1 -5.2  0.2  1.7
570 DATA 12.3  1.2  8.7 12.3 -4.7 -.1  3.2  2.1  4.3  1.8  1.9  2.3
580 DATA  5.7  4.7 -2.8  5.7  7.4 2.1 12.7  1.1 32.1  4.5  1.1  8.3
590 DATA  1.7 -6.7  5.6  7.4  1.2 3.5  2.7  2.8  1.9  1.7 -4.5 -9.9
600 DATA  3.1 -3.4 -9.2 10.6  8.9 1.7 -1.1 -0.3  3.2  3.2  1.6  1.3
610 DATA 13.3 -9.7  5.2  7.6 1.1 86.2  1.3  3.2  9.7  1.2 87.1 -9.2
620 DATA  1.2  3.4 -4.5 -6.7  9.8 0.1 -2.1  5.8 -9.1 -5.2  0.2  1.7
630 DATA 12.3  1.2 -8.7 12.3 -4.7 -.1 -3.2  1.8  1.9  2.3  3.1  4.3
```

Listing 5

Of course, the BASIC program might be recoded more efficiently, but the IFP program could also be recoded more efficiently. In the case of IFP, the recoding can be done rigorously via algebraic theorems. For example, the *Aik* function is called twice per iteration, once by *L* and once by *U*. If we define a new function *LU* and substitute the definitions of *L* and *U* we get:

```
DEF LU AS
   [
      IF Singleton THEN #<<1.0>>         (* definition of L *)
      ELSE
         [
            L11,
            Aik | [EACH #0 END,L] | apndl
         ] | ApndlCol
      END,

      IF Singleton THEN id               (* definition of U *)
      ELSE
         [
            U1k,
            Aik | [EACH #0 END,U] | ApndlCol
         ] | apndl
      END
   ];
```

Applying the algebraic rules, we finally get:

```
DEF LU AS
   IF Singleton THEN [#<<1.0>>,id]
   ELSE
      [
         L11,
         Aik|[EACH #0 END,LU],
         U1k
      ] |
      [
         [
            1,
            2 | [1,2|1] | apndl
         ] | ApndlCol,
         [
            3,
            2 | [1,2|2] | ApndlCol
```

```
                ] | apndl
            ]
        END;
```

which runs 1.5 times as quickly as the separate L and U functions.

References

[Aho84]
Alfred Aho, "Awk - A Pattern Scanning and Processing Language," pp. eter Weinberger in *Unix User's Manual - Supplementary Documents,* (March 1984).

[Bac78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebsa of Programs," *CACM* **21,8** pp. 613-641 ACM, (August 1978).

[Bac81]
John Backus, "The Algebra of Functional Programs: Functional Level Reasoning, Linear Equations, and Extended Definitions," in *Formalization of Programming Concepts,* Springer Verlag, New York (1981).

[Bac84]
John Backus, "Transforming Functional Programs," Lecture at University of Illinois at Urbana-Champaign (September 18, 1984).

[Bad83]
Scott Baden, "Berkeley FP User's Manual, Rev. 4.1," *UNIX Programmers Manual,* (July 27,1983).

[Dar82]
J. Darlington, J.V. Guttag, P. Henderson, J.H. Morris, J.E.Stoy, G.J. Sussman, P.C. Treleaven, D.A. Turner, J.H. Williams, and D.S. Wise, *Functional Programming and its Applications,* Cambridge University Press (1982).

[Fod83]
John K. Foderaro, Keith L . Sklower, and Kevin Layer, "The FRANZ LISP Manual," in *UNIX Programmer's Manual - Supplementary Documents,* (June 1983).

[Har85]
Peter G. Harrison and Hessam Khoshnevisan, "Functional Programming Using FP," *BYTE* **10,8** pp. 219-232 (August 1985).

[Ive62]
Kenneth Iverson, *A Programming Language,* Wiley, New York (1962).

[Kel86]
Robert M. Keller and M. Ronan Sleep, "Applicative Caching," *ACM Transactions on Programming Languages and Systems* **8,1** pp. 88-108 ACM, (January 1986).

[Ker78]
Brian W. Kernighan and Dennis M . Ritchie, *The C Programming Language,* Prentice-Hall Inc., Englewood Cliffs, New Jersey (1978).

[Loe81]
R. G. Loeliger, *Threaded Interpretive Languages,* BYTE Publications, Peterborough NH (1981).

[Ran43]
Ayn Rand, *The Fountainhead,* Bobbs-Merrill Company, Inc., Indianapolis, New York (1943).

[Rob85]
Arch D. Robison, "IFP User's Manual," Professional Workstation Research Group Technical Report #7, University of Illinois, Urbana-Champaign (1985).

[Suz80]
Norihisa Suzuki, "Analysis of Pointer Rotation," *Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 1-11 ACM, (January 1980).

[Ten76]
R. D. Tennent, "The Denotational Semantics of Programming Languages," *CACM* **19,8** pp. 437-453 (August 1976).

[Wir77]
Niklaus Wirth, "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?," *CACM* **20,11** pp. 822-823 ACM, (November 1977).

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1327	2.	3. Recipient's Accession No.
4. Title and Subtitle A FUNCTIONAL PROGRAMMING INTERPRETER			5. Report Date March 1987
			6.
7. Author(s) Arch Douglas Robison			8. Performing Organization Rept. No. UIUCDCS-R-87-1327
9. Performing Organization Name and Address Department of Computer Science 1304 W. Springfield Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NASA NAG 1-613
12. Sponsoring Organization Name and Address NASA Langley Research Center Hampton, VA 23665			13. Type of Report & Period Covered M. S. Thesis
			14.

15. Supplementary Notes

16. Abstracts

Functional Programming (FP)[Bac78] is an alternative to conventional imperative programming languages. This thesis describes an FP interpreter implementation. Superficially, FP appears to be a simple, but very ineffecient language. Its simplicity, however, allows it to be interpreted quickly. Much of the inefficiency can be removed by simple interpreter techniques.

This thesis details the design of the Illinois Functional Programming (IFP) interpreter, an interactive functional programming implementation which runs under both MS-DOS and UNIX. The IFP interpreter allows functions to be created, executed, and debugged in an environment very similar to UNIX. IFP's speed is competitive with other interpreted languages such as BASIC.

17. Key Words and Document Analysis. 17a. Descriptors

functional programming
programming language
parallel processing
compiler

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 66
	20. Security Class (This Page UNCLASSIFIED	22. Price